

Whiley Cheat Sheet

By David J. Pearce, 2014. See <http://whiley.org>

Values

Values are the fundamental units of execution in Whiley and have value semantics, rather than reference semantics (as in many object-oriented languages).

<code>null</code>	<i>Null value</i>
<code>true</code> <code>false</code>	<i>Boolean values</i>
<code>123</code> <code>-99</code> <code>0xFF</code>	<i>Integer values</i>
<code>"Hello"</code> <code>"new\n_line"</code>	<i>String values</i>
<code>(1,2,3)</code> <code>(true,null)</code>	<i>Tuple values</i>
<code>[1,2,3]</code> <code>[1, "xyz"]</code>	<i>Array values</i>
<code>{name: "dave"}</code> <code>{x: 1, y: 0}</code>	<i>Record values</i>

Types

The Whiley programming language is *statically typed*, meaning that every expression has a type determined at compile time. Furthermore, evaluating an expression is guaranteed to yield a value of its type.

<code>null</code> <code>bool</code> <code>int</code>	<i>Primitive types</i>
<code>int null</code> <code>bool int</code>	<i>Union types</i>
<code>(int, int)</code> <code>(int, null, bool)</code>	<i>Tuple types</i>
<code>int []</code> <code>bool [] []</code> <code>(int null) []</code>	<i>Array types</i>
<code>{bool f}</code> <code>{int len, int [] is}</code>	<i>Record types</i>

Expressions

The majority of work performed by a Whiley program is through the execution of *expressions*. Every expression produces a value and may have additional side effects.

<code>x + 1</code> <code>2 * y</code> <code>z - 1</code> <code>(x + y) / 2</code>	<i>Arithmetic</i>
<code>x < y</code> <code>0 >= z</code> <code>x == y</code> <code>x != y</code>	<i>Comparisons</i>
<code>!x</code> <code>x & y</code> <code>x y</code> <code>x ==> y</code> <code>x <==> y</code>	<i>Boolean</i>
<code> ls </code> <code>ls[0]</code> <code>[true; n]</code>	<i>Arrays</i>
<code>[1, x+y]</code> <code>x in xs</code>	<i>Records</i>
<code>{x: 1+y}</code> <code>xr.f</code> <code>xr.f.g</code>	<i>Records</i>
<code>all { i in 0.. xs xs[i] >= 0 }</code>	<i>Quantifiers</i>
<code>no { i in 0.. xs xs[i] < 0 }</code>	
<code>some { i in 0.. xs xs[i] >= 0 }</code>	
<code>x is null</code> <code>x is int</code>	<i>Type Tests</i>

Statements

The execution of a Whiley program is controlled by *statements*, which cause effects on the environment. Statements in Whiley do not produce values. Compound statements may contain other statements.

Variables are declared and initialised through *variable declarations*. Variables must be declared before being used.

<code>int x</code>	<code>int x = 1</code>	<code>int x, int y</code>
--------------------	------------------------	---------------------------

Variables, fields and map or list elements can be *assigned*. Variables must be defined before being used.

<code>x = x + y</code>	<code>x[0] = 1</code>	<code>r.f = 3</code>	<code>x, y = t</code>
------------------------	-----------------------	----------------------	-----------------------

Conditional statements control the flow of execution based on the result of a boolean expression.

<code>if x < 0:</code>	<code>if x < 0: ... else: ...</code>	<code>if x < 0: ... else if x > 0: ...</code>
---	---	---

Looping statements control the flow of execution by repeating some sequence of statements zero or more times.

<code>while x < 0:</code>	<code>do: ... while x < 0 ...</code>
--	---

Switch statements control execution flow by matching the result of an expression.

<code>switch x: case 1: x = x + 1 case 1, 2: x = 0 ...</code>	<code>switch x: case 1: x = x + 1 default: x = 0 ...</code>
---	---

Return statements terminate the execution of a function or method and may return the result of an expression.

<code>return</code>	<code>return x + 3</code>	<code>return x, y</code>
---------------------	---------------------------	--------------------------

Assertion and assumption statements enable the programmer to express knowledge at a given point.

<code>assert x > 0</code>	<code>assume x > 0 ==> y < 3</code>
------------------------------	--

Break statements terminate loops early; *debug* statements enable output from functions; *skip* statements are a no-op.

<code>break</code>	<code>debug "got_here"</code>	<code>skip</code>
--------------------	-------------------------------	-------------------

Declarations

A *declaration* declares a named entity within a source file and may refer to named entities in this or other source files and (in some cases) may *recursively* refer to itself.

Constant declarations define constants with known values at compile-time (they cannot be recursively defined).

```
constant TEN is 10
constant TWENTY is TEN * 2
```

Type declarations define named types composed from other types (they may be recursively defined).

```
type Point is { int x, int y }
```

```
type Link is { LinkedList next, int data }
type LinkedList is null | Link
```

Function declarations define functions which are *pure* and may not have side-effects. They are guaranteed to return the same result given the same arguments, and are permitted within specifications.

```
function find(int[] xs, int x) -> int:
  ...
```

Method declarations define methods which are *impure* and may have side-effects. They cannot be used within specifications.

```
method main(System.Console console):
  console.out.println("Hello_World")
```

Specifications

A *precondition* is a condition over the parameters of a function that must hold when the function is called. A *postcondition* is a condition over the return values of a function which is required to be true after the function is called.

```
function decrement(int x) -> (int y)
// Parameter x must be greater than zero
requires x > 0
// Return must be greater or equal to zero
ensures y >= 0
// Return must be less than input
ensures y > x:
  //
  return x - 1
```

A *data-type invariant* is a constraint on the values of a declared type which must be true for any instance of it.

```
type nat is (int n) where n >= 0
type pos is (int p) where p > 0
```

A *loop invariant* is a property which holds before and after each iteration of the loop, such that: **(1)** the loop invariant must hold on entry to the loop; **(2)** assuming the loop invariant holds at the start of the loop body (along with the condition), it must hold at the end; **(3)** the loop invariant (along with the negated condition) can be assumed to hold immediately after the loop.

```
...
int i = 0
while i < x where i >= 0:
  i = i + 1
...
```

Examples

The following function computes the maximum value of two integer parameters.

```
function max(int x, int y) -> (int z)
// must return either x or y
ensures x == z || y == z
// return must be as large as x and y
ensures x <= z && y <= z:
  // implementation
  if x > y:
    return x
  else:
    return y
```

The following function uses a **break** to exit a **while** loop when the first element matching parameter *x* is found.

```
// Find index of matching element, or return -1
function indexOf(int[] xs, int x) -> int:
  int i = 0
  //
  while i < |xs| where i >= 0:
    if xs[i] == x:
      return i
    i = i + 1
  return -1
```

The following function computes the length of a linked list.

```
// A linked list is either the empty list or a link
type LinkedList is null | Link
// A single link in a linked list
type Link is {int data, LinkedList next}

// Return length of linked list (i.e. number of links it contains)
function length(LinkedList l) -> int:
  if l is null:
    // l now has type null
    return 0
  else:
    // l now has type {int data, LinkedList next}
    return 1 + length(l.next)
```

The following function reverses the values in a list of integers.

```
function reverse(int[] xs) -> (int[] ys)
// size of lists are the same
ensures |xs| == |ys|:
  int i = 0
  int[] zs = xs
  //
  while i < |xs| where i >= 0 && |xs| == |zs|:
    int j = |xs| - (i+1)
    xs[i] = zs[j]
    i = i + 1
  return xs
```